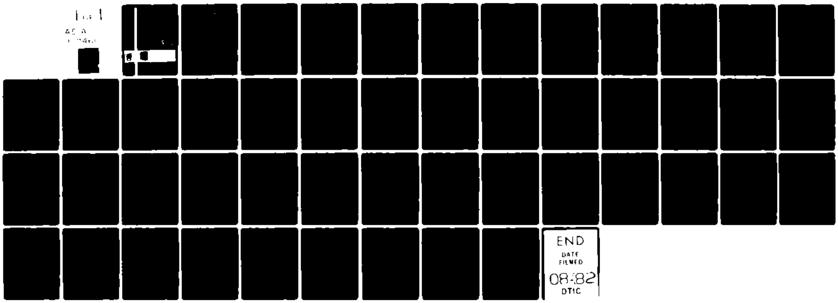
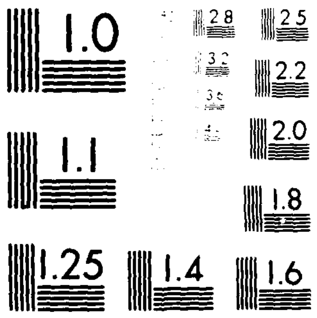
AD-A117 461 ARIZONA UNIV TUCSON ENGINEERING EXPERIMENT STATION F/G 9/2
A SUMMARY DISCUSSION OF CONLAN.(U)
JUL 82 F J HILL N00014-79-C-0368
UNCLASSIFIED NL



END
DATE
FILMED
08-82
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

(12)

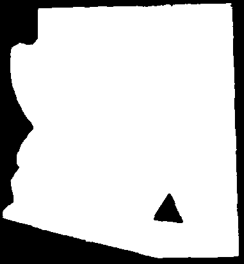
A Summary Discussion of CONLAN

Fredrick J. Hill
Dept. of Electrical Engineering
University of Arizona
Tucson, Arizona 85721

15 July 1982
Final Report N00014-79-C-0368

Prepared for

Office of Naval Research
ONR:437:MD:lbp
NR 048-656
Arlington, Virginia 22217



ENGINEERING EXPERIMENT STATION
COLLEGE OF ENGINEERING
THE UNIVERSITY OF ARIZONA
TUCSON, ARIZONA 85721

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A117461	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Summary Discussion of CONLAN		5. TYPE OF REPORT & PERIOD COVERED FINAL
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Fredrick J. Hill		8. CONTRACT OR GRANT NUMBER(s) N00014-79C-0368
9. PERFORMING ORGANIZATION NAME AND ADDRESS Dept. of Electrical Engineering University of Arizona Tucson, Arizona 85721		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of NAVAL Research ONR;437:MD:1bp NR 048-656 Arlington, Virginia 22217		12. REPORT DATE July 15, 1982
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

UNLIMITED

17. DISTRIBUTION ST. [illegible]ENT (of [illegible] abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY [illegible]TES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

hardware description language, standard, VLSI, languages, CAD,

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

CONLAN is a proposed family of standard hardware description languages developed by a six person international working group. It stresses strong typing and formal semantic development so that the results of simulation at different language levels can be formally checked. This project funded a small portion of the CONLAN development including the writing of the CONLAN introduction contained herein. This introduction will be adapted to form the first chapter of the overall CONLAN report to be published by Springer-Verlaag,

DD FORM 1 JAN 73 1473

UNCLASSIFIED

Project Summary

CONLAN is a proposed family of standard hardware description languages developed by a six person international working group. It stresses strong typing and formal semantic development so that the results of simulation at different language levels can be formally checked. This project funded a small portion of the CONLAN development including the writing of the CONLAN introduction contained herein. This introduction will be adapted to form the first chapter of the overall CONLAN report to be published by Springer-Verlaag. All rights relating to republication of this material are reserved.

CONLAN is the result of numerous meetings in several countries and considerable individual effort at the institutions of the working group members. Professor Robert Piloty of the Technical University of Darmstadt has served as chairman of the working group. His inspiration determined the major direction of CONLAN. Some of the most important sections of the CONLAN draft report have been included as appendices.

Contents

List of Figures

1.1 An Introduction to CONLAN

CONLAN is short for Consensus Language, more specifically, a consensus hardware description language. This report is intended to serve as a self-contained complete description of CONLAN as it stands at the time of publication. The purpose of this part of this report is to provide an informal introduction to CONLAN. Reading part I first will enhance both understanding and appreciation of the formal presentation in parts II and III which set forth the semantics and syntax of the CONLAN language, respectively.

1.1.1. Historical Introduction

The CONLAN project began in 1973 as an attempt to consolidate existing hardware description languages into a standard language. To this end a series of mail ballots were distributed and collected by Dr. G. J. Lipovski, chairman of the Conference on Digital Hardware Languages (CDHL). In September, 1975, a new approach began with the formation of a working group consisting of Dominique Borrione, Yaohan Chu, Donald Dietmeyer, Fredrick Hill, Patrick Skelly, and chaired by Robert Piloty. The Working Group has met at approximately six month intervals in Ottawa, Valley Forge, Toronto, Tucson, Huntsville, Grenoble, Pittsburgh, Palo Alto, Darmstadt, and Anaheim. Support has been provided by Bell Northern Research, Sperry Univac, Office of Naval Research, Ballistic Missile Defense Advanced Technical Center, Institut National pour la Recherche en Informatique et Automatique, Bundesministerium fur Forschung und Technologie, Siemens, and Fujitsu. Additional support was provided by the author's institutions. After the first three meetings, Professor Chu was unable to continue and was replaced by Mario Barbacci.

1.1.2. Motivation and Objectives

The decision to start the CONLAN project was motivated by the following assessment of the situation in the area of HDL and of Computer Aided Design (CAD) tools based on them. Several dozens HDL's existed in 1973 and every year since new languages have been proposed and published, mostly from persons in academic institutions. This tendency to proliferation is in sharp contrast to acceptance in industry. Neither have they been used to document the design process of digital systems nor to support tools for certification, synthesis, and performance evaluation to any appreciable extent. Most CAD tools in industry are designed to aid the manufacturing process (placement, routing, mask layout etc.). The process of systems and logic design is mostly carried out in the traditional way of drawing block and circuit diagrams at the IC package or gate level. In many cases these diagrams are the only true and complete documentation of the system. Most other aspects or phases of the system design, particularly system behaviour, are informally and incompletely described. Simulation as a means for advanced certification is used, if at all, at a very low level (mostly gate level) and hence at enormous cost for more complex systems. Most of the certification is done very late at the level of a physical prototype causing costly changes in physical design.

This situation has not changed very much in the five years of CONLAN development: HDL's continue to proliferate, but their usage in real life design has not increased in the same proportion. Only recently a growing interest in efficient tools for design support at systems and logic level can be observed, probably due to the advance of LSI and VLSI, where late changes make a system more and more costly, and due to increased system complextity in a competive market, calling for more efficient design tools.

There are several reasons why acceptance of existing HDL's is so low:

1. None of the languages alone is of sufficient scope to portray all aspects of a system and cover all phases of the design process.
2. Languages of different scope are syntactically and semantically unrelated.
3. Few of the languages are formally defined.
4. Only a few languages are implemented.
5. Descriptions are represented by character strings rather than diagrams.
6. There exists no comprehensive hardware and firmware design methodology telling how to use HDL's effectively.

The main aim of the CONLAN Working Group is to remedy the first four deficiencies. Its primary objectives are:

1. To provide a common formal syntactic and semantic base for all levels and aspects of hardware and firmware description, in particular, for descriptions of system structure and behavior.
2. To provide a means for the derivation of user languages from this common base:
   a. having a limited scope adjusted to a particular class of design tasks,
   b. thus being easy to learn and simple to handle,
   c. yet having a well-defined semantic relation among each other.
3. To support CAD - tools for documentation, certification, design space exploration, synthesis and so on.
4. To avoid imposing a single rigid style of hardware description on makers of design tools.

The above objectives for a consensus hardware description language call for a language capable of representing hardware at several distinct levels of abstraction. At the lowest level, in which even flip-flops are nonprimitive, gate networks must be described in CONLAN. In contrast, algorithms provide very abstract models of hardware that must interact with lower level hardware descriptions. Thus, these higher level representations must also be expressible in CONLAN. Since the number of levels and the boundaries of each are not universally accepted, CONLAN must be viable for any reasonable set of boundaries.

1.1.3. The CONLAN Approach

The requirements of this range of language levels would seem to suggest not a single consensus language but a family of related languages, languages with consistent syntax and semantics for similar object types and operations. The need for a family of languages requires that CONLAN be an extensible family since all members cannot be prepared simultaneously, indeed all potential members cannot be envisioned at any point in time. That family members be consistent requires that extensibility be controlled.

CONLAN addresses these fundamental requirements by supporting a self-defining, extensible family of languages. Its member languages are tied together by a common core syntax and a common semantic definition system. The CONLAN construct to define a member language is called a **language definition segment**. Each new language definition segment is based upon an existing one, its reference **language**, to enhance consistency and increase language definition efficiency. Member languages are also used to write descriptions of hardware, firmware or software modules, of course. **Description definition segments** are provided for this purpose.

The CONLAN family of languages is open ended. New languages may be derived from existing ones at any given point in time as the need arises. These in turn may be used later as reference languages for further languages.

The same construction mechanism and the same notational system used to provide descriptions is also used to define new language members. In this sense CONLAN is self-defining in contrast to externally defined languages using a separate language to define its semantics, e.g. the formal description of PL/I using the Vienna Definition Language.

To initialize the CONLAN family the CONLAN Working Group has prepared the root language called **Base CONLAN** (bcl) as the interface between the CONLAN Working Group and its public. Bcl provides a carefully chosen set of basic objects reflecting the CONLAN concept of time and space, of signals and carriers, of arrays and records. These concepts are expected to prevail throughout the family. To test and illustrate the language construction mechanism a very low level but powerful language called **primitive set conlan** (pscl) was used to formally define bcl. Pscl has no parent language. It owns a set of primitive objects whose domains and operations are introduced informally.

Base CONLAN is primarily a starting point, with well defined and semantically sound primitives, for language designers to derive a coherent and comprehensive family of computer hardware description languages. As a result, hardware descriptions written in Base CONLAN may look verbose. Nevertheless, all concepts pertinent to hardware modeling are already in Base CONLAN and may be common to all languages of the CONLAN family.

1.2 CONLAN Concepts

The purpose of this informal narrative is to prepare the reader for the formal portions of this report and to focus attention on the more significant aspects of CONLAN. The report itself is the complete definition of CONLAN and no attempt will be made to repeat its exhaustive treatment. In fact, details and examples will be avoided here. Many concepts and notational conventions of existing programming and hardware description languages will be found in CONLAN. To some extent decisions to include or exclude such concepts and conventions will be justified; clearly all details of five years of group research and debate cannot and should not be fully reported.

The well known advantages of structured programming in hardware description languages justify incorporation of the concept in CONLAN. Valid CONLAN text consists of properly nested blocks. Major blocks which define an entity, either a language item or a hardware module, are called **segments**. Blocks that refer to or invoke an instance of a segment are called **statements**. Bodies of segments typically consist of statements. Segments begin with a keyword followed by an identifier and terminate with keyword END which may, but need not be followed by the segment identifier. Most statements begin with a keyword and terminate with END which may be followed by the opening keyword. These concepts are not new; their familiarity should put the reader at ease.

1.2.1. Language and Description Definition Segments

We begin exploring the CONLAN segments with the types of segments previously mentioned. The language definition segment (keyword CONLAN) defines a new member of the CONLAN family. The description definition segment (keyword DESCRIPTION) defines a hardware or firmware system. To a large extent different groups of people are expected to write these two types of segments. Those who prepare new members of the CONLAN family and software to support their new members will be referred to as **toolmakers**; those who use those new languages and supporting software to record their hardware design efforts are **users**. In many organizations toolmakers are members of the Design Automation Department and users are members of Engineering Design Departments. A toolmaker writes a CONLAN segment; a user writes a DESCRIPTION. The outermost segment in both cases must be prefaced with a reference language statement (keyword REFLAN). The existing (reference) language revealed in the REFLAN statement provides the syntax and semantics immediately available for use in writing the body of the outermost segment. A new language is said to be **derived** from its reference language. All members of the CONLAN family are derived from bcl, perhaps through a long chain of intermediate languages.

Several methods for hiding segments of a language from users and future toolmakers are available in CONLAN. First, segments marked PRIVATE are neither visible nor accessible to future toolmakers and users. Second, identifiers and keywords that terminate with the symbol @ are not visible to users. Hiding and preventing language features from appearing in derived languages may also be accomplished by syntax modification statements (keyword FORMAT) which will be discussed in a subsequent section. These hiding mechanisms permit toolmakers to prepare simple hardware description languages for specific application areas while maintaining a clear semantic and syntatic relation with their ancestor languages.

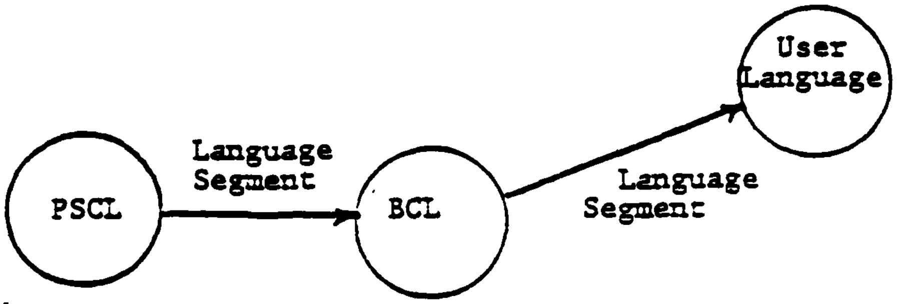
While nonPRIVATE segments of a reference language provide objects and operations for a toolmaker to use in writing a CONLAN segment, those objects and operations are not automatically available to users of the new language and future toolmakers. Redefinition of segments to be propagated from the reference to the new language is avoided by the CARRY statement in which segments to be propagated are revealed. Keyword CARRYALL may replace the CARRY statement in a special, obvious case. Further, EXTERNAL statements may be written in DESCRIPTION as well as CONLAN segments to avoid repetitive writing of the same CONLAN text. EXTERNAL statements stand in place of the original segment definition, just as though that original segment were copied at the point of the EXTERNAL statement.

To summarize, the CONLAN text structure is shown in Figure 1. At any given point in time it consists of a set of language definition segments and a set of description segments. Each segment is under the scope of a REFLAN statement. In a language definition segment this statement points to the language from which the new language is directly derived. Thus language LL1 is directly derived from L1, and LL2 is directly derived from L2. In a description segment the REFLAN statement points to the language in which the description is written.

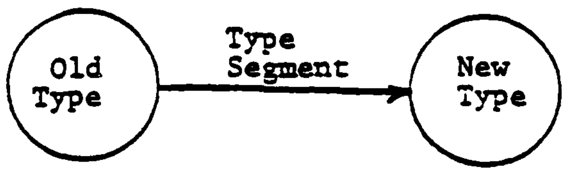
Subsequent sections of this part of the report offer more detailed information on CONLAN and DESCRIPTION segments.

1.2.2. Type and Operation Definition Segments

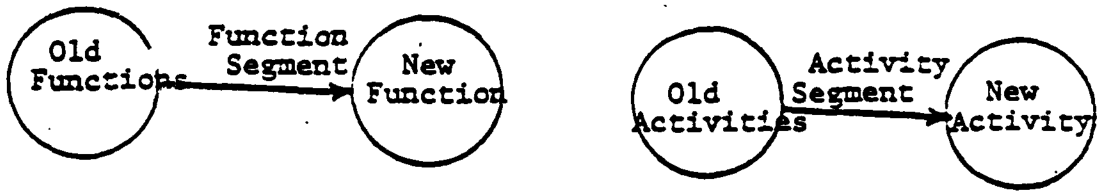
Objects are the things with which CONLAN is concerned. Pscl provides the fundamental objects of the CONLAN family as we will see. Bcl provides more elaborate objects that are expected to be useful for hardware description throughout the CONLAN family. Other members of the CONLAN family are expected to provide additional hardware and firmware items as objects. Objects are usually collected into sets that may or may not be explicitly named. All objects are members of at least one named set provided by pscl. The CONLAN method for language construction and hence semantic definition is based upon the concept of abstract data types as developed for programming languages CLU [Liskov, ?] and ALPHARD[Wulf, ?]. A CONLAN abstract data type, or simply type, consists of a set of objects together with operations defined on the members of that set. The TYPE segment provides a means for specifying the set membership and defining the operations of the defined type. The identifier of the TYPE segment is used as the name of the set of objects, but this should not lead one to think of a type as just a set of objects. The identifier of a TYPE segment may be parameterized. Such definitions provide a family of types. A member of the family is obtained by binding the formal parameters to actual parameters.



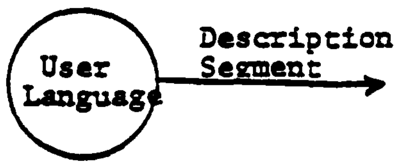
(a.)



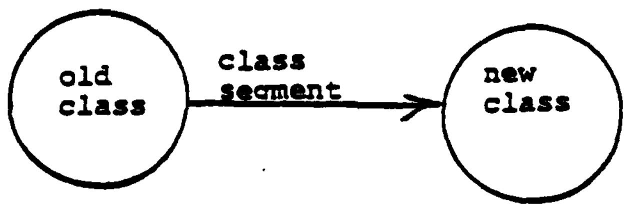
(b.)



(c.)



(d.)



(e)

Figure 1.2-1 Segments

As may be expected, TYPE segments within a CONLAN or DESCRIPTION segment may be written in terms of the nonPRIVATE types of the reference language. In addition, TYPE segments may be written in terms of previously written TYPE segments. This leads to a partial order on the types of a language, just as the reference language provides a partial order on the members of the CONLAN family.

Experience with programming languages has revealed the value of requiring users of a language to specify the set of which each referenced object is a member, i.e., to specify the type of all identifiers used. While hardware description languages have not provided traditional data types (integer, real, complex, etc.), many have introduced hardware types (register, terminal, clock, etc.). To gain error checking capability CONLAN requires that the type of all objects be specified.

When all operations on an existing type are desired on a subset of that type or a member of a type family, repetitive definition is avoided with the concise SUBTYPE segment which permits no new operations to be introduced.

On occassion it is necessary for toolmakers to group together types with a common property. A set of such types together with operations defined on members of that set is a class. Members of a class are thus types, and not objects of the grouped types. One class that contains all types is provided by pscl. A new class is defined with the CLASS segment.

A great deal of time and effort has gone into precisely defining many things that are often taken as "known" in programming and hardware description languages. "Operation" is an example. In summary here, CONLAN admits two types of operations. First, an operation is a formal rule for selecting one or more "result" objects from designated result types. This corresponds to the mathematical function. Second, an operation is the selection of one or more result objects and their placement in container objects. Objects that are able to hold other objects are known as carriers. Pscl provides the basic carrier and operations on that carrier, i.e. the basic carrier type.

New operations are defined with FUNCTION and ACTIVITY segments. The FUNCTION segment defines an operation that selects and returns an element of a specified type and has no side effects, as might be expected. "Return" is another of those "known" concepts: Fortunately the CONLAN definition is consistent with the usual notion. The ACTIVITY segment defines an operation that has side effects in that the contents of one or more carriers named in the formal parameter list of the segment are modified. An ACTIVITY does not return objects. Thus identifiers of ACTIVITY segments may not appear in expressions. Activities parallel procedures and subroutines of programming languages in this respect.

CONLAN text, like text of all formal languages, has significance only if it is read and responded to by a person or machine (an environment) and then only to the extent that the environment reads and responds in a well defined manner. Many pages of this report are devoted to specifying the manner in which CONLAN text is to be read if logic simulation is intended. But not all such rules can be written in advance for all future members of the CONLAN family, not all applications of CONLAN text can be anticipated, and some responses are not appropriately specified via CONLAN text but best left to software toolmakers, e.g., response to error conditions. FUNCTION and ACTIVITY segments may be prefixed with keyword INTERPRETER@ to indicate that the defined operations are to be invoked by the environment of the CONLAN member of which the operations are a part. The environment is expected to call upon such operations appropriately; this is the responsibility of the software toolmaker. Users may not write such operations, of course, and should seldom be aware that they exist in the CONLAN member languages they use.

1.2.3. Identifiers, Parameters and Interface Carriers

CONLAN utilizes the international ISO 646-IRV character set and code. The semantics of most symbols of that set have been determined by the Working Group. Toolmakers may assign semantics to a few remaining symbols and to concatenations of symbols (compound symbols). Members of the upper case are called capitals; members of the lower case are letters. It should be obvious by now that keywords are sequences of capitals possibly terminated by symbol @. CONLAN text destined for publication where a richer character set is available may utilize a limited set of specified substitutions for the ISO 646-IRV characters, and different typefaces.

CONLAN deals with the following categories of objects:

- Elements
- Parameters
- Operations
- Types and Classes
- Descriptions
- Languages

Identifiers are most often used to denote CONLAN objects. **Simple identifiers** start with at least one letter followed by an arbitrary string of letters, digits, and underscore ('_'), of any length and terminating with a letter, or digit, or with the symbol @. While software may impose a length limitation on identifiers, in principle identifiers may be very descriptive words or sequences of words separated by underscores. Contiguous underscores are prohibited in CONLAN to encourage this, their intended use. Symbol @ denotes a system identifier which may be used only within a language definition segment. **Compound identifiers** are two or more simple identifiers separated by periods ('.'). They permit one to prefix a simple identifier with one or more enclosing segment identifiers when the corresponding object is referenced outside the segment providing the simple identifier.

Elements are the operands for CONLAN operations. While identifiers may always stand in their place, constant denotations are available for identifying specific members of many CONLAN types. We will see constant denotations when we preview pscl.

Formal and actual parameters in CONLAN are as in most languages. A formal parameter is denoted by an identifier representing any element of a given type. Formal parameters appearing in a parameter list of an operation, type or description must be typed i.e. the parameter must be followed by the identifier of the type of which the parameter is a member separated by a colon (e.g. x:bool). The type identifier (bool) defines the domain of x and the operations applicable to it. This requirement facilitates checking: When a formal parameter is bound to an actual parameter, its associated type designator is used to check if the type of the actual parameter is equivalent to the type of the formal parameter. The type of the actual parameter is normally explicitly stated in a declaration, or in one of the constructs involving predicates (yet to be discussed). If the actual parameter is a constant denotation, it can also be checked to determine that it belongs to the domain of the prescribed type. The binding of actual to formal parameters and the importing and exporting of information from segments are discussed in great detail in part II.

CONLAN operations are normally denoted by an identifier prefixing a list of parameters (operands). Operator symbols may be introduced via syntax modification for prefix and infix notation either in lieu of a standard functional notation or as an alternative to it. The typed identifiers enclosed in parentheses following the identifier of a DESCRIPTION segment are seldom parameters in the conventional sense. Attributes may appear; if they do a family of descriptions is defined. Such parameters are treated quite differently from the other entries, the interface carriers. Interface carriers must be carrier objects. In addition to being typed, interface carriers are tagged to indicate the direction of information flow that they support. They provide all communication between the inside of the description and its environment, just as all communication with an encapsulated hardware module (an IC) must be via its pins. We begin to see a fundamental difference between operations and descriptions. Generally, operations express behavior; descriptions express structure as well as behavior. This difference will be more fully explored later when invocation of operations and instanciation of descriptions is discussed.

1.2.4. Statements

While the intent of the CARRY, EXTERNAL and REFLAN statements has already been revealed, many CONLAN statements remain to be discussed. A predicate is any expression that produces a boolean result. The first statements to be discussed are forms of predicates. FORALL@, FORSOME@ and FORONE@ statements test a specified set of objects against an embedded predicate and evaluate to true (1 in CONLAN) or false (0) as the keywords suggest. These statements were particularily valuable in building bcl from pscl; their use in building hardware description languages from bcl is expected to be much lower.

The THE@ statement selects the one member of a revealed set for which an embedded predicate is true. The ALL statement selects all members for which an embedded predicate is true; it constructs a subset and is particularily useful in writing TYPE segments. Since users may have occassion to write such segments the ALL keyword is not restricted by a terminal @ symbol.

The IF and CASE statements selectively invoke one of a list of operations based upon the value of a predicate or predicates. Both permit an ELSE part. The IF statement also permits an unlimited number of ELSEIF parts.

The OVER statement provides a short way of writing a collection of operation invocations that differ only in the value of an index. The OVER statement is not a loop constructor as found in programming languages where sequence as well as invocation is specified. In fact, provisions for expressing sequential invocation are largely absent from pscl and bcl.

Expressions express a sequence in which functions are to be invoked via an operator hierarchy and the use of parentheses. But procedural descriptions of hardware are not possible until a CONLAN member language is written that provides suitable control structures. The Working Group has performed experiments and believes that bcl supports the development of such control structures, and therefore expects to see them in the first CONLAN members derived from bcl.

The DECLARE statement provides a means of assigning a name to an element of a specified type or of renaming an already named object. In hardware descriptions the local carriers must be declared as we will see. The importance of being able to assign multiple names (naming a subregister) is well known. Further, using names throughout a program or hardware description rather than constant denotations is considered to be good practice. Contrast the DECLARE with the USE statement which assigns a name to an instance of a description and implicitly declares the interface carriers of that instance.

The DESCRIPTION segment may be thought to bring an infinite set of identical hardware units into the stockroom. No units exist in a specific system until one goes to the stockroom, requests a specific number of units, and assigns a name to each via a USE statement. When the units are acquired, their interface terminals come with them, and may be identified by a compound name involving the name assigned to the unit of which they are a part. We see more clearly the CONLAN distinction between an operation and a description.

The FORMAT@ statement provides the toolmaker with a means of deriving the syntax of a new language from the syntax of its reference language. Productions may be modified and given additional semantic meaning with the FORMAT@ statement, but not without limit. A system of tags on productions establishes the limits. The presence of one type of tag indicates that the production or alternative can be removed in a future CONLAN segment, but the *name of the non-terminal cannot be used for a new production*, i.e., it can be deleted but not replaced. The presence of a second type of tag on a production indicates that alternatives may be added to the production. The absence of all such tags indicates that the production or alternative may not be altered in any way. By carefully placing tags on the bcl grammar the Working Group intends to ensure the consistency of all CONLAN member languages while permitting toolmakers to keep the syntax of a new language as simple as possible and yet incorporate new constructs to denote specific features such as an infix symbol for a new, important operation. The set of productions that may never be deleted, and thus is common to all CONLAN members is the core syntax.

Finally, the ASSERT statement provides a means for toolmakers and users to specify predicates that they expect to be true. An error condition exists if an assertion proves to be false. While ASSERT statements have not been used extensively in developing bcl from pscl, they are expected to be especially valuable in hardware description. Setup and hold time specifications on flip-flops are naturally checked via assertions, for example.

1.2.5. Segment Templates

We are now in a position to appreciate a more detailed look at the CONLAN segments. In the templates given below a '#' is used to mark optional components and a '*' marks those parts that may appear in any order that does not reference a yet to be defined object. Forward references are not permitted in CONLAN. A great deal of flexibility as to the inclusion and order of inclusion of parts of segments will be noted.

The CONLAN segment has the form:

```
reference_language
CONLAN
name
BODY
carry_list #
type_definitions #*
subtype_definitions #*
class_definitions #*
operation_definitions #*
description_definitions #*
format_statements #*
ENDname
```

It is interesting to note that in CONLAN and DESCRIPTION segments operations may be defined outside of TYPE segments (floating operations), and that DESCRIPTION segments may be nested within a CONLAN segment. Thus a language may provide a stockroom of integrated circuits, for example. Also note in the description template below that DESCRIPTION segments may be nested without limit. Thus a description that models a hardware system need not be altered when that system suddenly becomes a component of a still larger system. And CONLAN provides maximum flexibility for stepwise refinement of a design. Instances of purely behavioral descriptions can be interconnected with instances of detailed network descriptions. Moreover, inside the same DESCRIPTION segment, some parts may be expressed in terms of clearly identified hardware elements, while other parts are described abstractly.

A reference language statement is required only on the outermost DESCRIPTION.

```
reference_language #
DESCRIPTION
name
(attributes) #
(interface_list) #
assertions #
BODY
assertions #*
type_definitions #*
subtype_definitions #*
class_definitions #*
operation_definitions #*
description_definitions #*
element_declarations #*
description_uses #
operation_invocations
ENDname
```

One or more operations must be invoked in a description; a description must do something, it must have some behavior. CONLAN does not support purely structural descriptions of hardware.

The TYPE segment template is:

```
TYPE
name
(parameters) #
BODY
set_definition
carry_list #
type_definitions #*
subtype_definitions #*
class_definitions #*
operation_definitions #*
constant_denotations #
ENDname
```

The CLASS segment template is very similar. Perhaps the most surprising thing revealed by the TYPE template is that local types and classes are permitted.

Operations are defined with the following templates:

```
reference_language #                    reference_language #
FUNCTION                                ACTIVITY
name                                    name
(parameters) #: type_designator         (parameters) #
assertions #                            assertions #
BODY                                    BODY
```

```
assertions #*                               assertions #*
type_definitions #*                         type_definitions #*
subtype_definitions #*                      subtype_definitions #*
class_definitions #*                        class_definitions #*
operation_definitions #*                    operation_definitions #*
element_declarations #*                     element_declarations #*
operation_invocations #                     operation_invocations #
RETURN result_expression
format_statements #                         format_statements #
ENDname                                     ENDname
```

The @ on keyword FORMAT@ prevents users from including such statements in their operation definition segments. But toolmakers will use FORMAT statements to extend their grammars to recognize invocations of the new operation, perhaps with infix symbolism, ,and extend the semantics of existing productions when the new operator on a new type closely parallels operations on older types.

1.3 Primitive Set CONLAN (pscl) Preview

CONLAN is built from the primitive set language pscl. Pscl is an exceptional member of the CONLAN family in that there is no reference language for pscl, object sets are assumed rather than defined, and operators on objects are assumed rather than being defined in more fundamental terms. The types and operations which are assumed in pscl are previewed in this chapter.

Type **univ@** consists of all members of all types defined in any member of the CONLAN family, together with operators '=' and '~='. It permits the present definition of operations on objects to be defined in the future, and is a necessary part of pscl if highly exceptional rules are to be avoided in CONLAN. Type univ@ is considered as the defining type for the other types of pscl, from which all other types of CONLAN will be derived.

Class **any@** is the universal class in CONLAN, and the only class in pscl. Its domain is the set of all types defined in any member of the CONLAN family, together with operations '=', '~=', '.<', and '<|'. Function '.<' may be used to determine if an object from univ@ is a member of a defined type (a member of any@). Function '<|' may be used to determine if a member of any@ (a type) was derived from another member of any@.

Figure 2 may clarify the universal set univ@ and the universal class any@. Since univ@ is a type, it is a member of any@. Those subsets of univ@ that are named via a type definition are also members of any@. Unnamed sets may be created in CONLAN by enumeration; they are not members of any@. Univ@ and any@ have the same significance to CONLAN as a universal set has to set algebra. They are necessary to place CONLAN on a sound theoretical foundation.

Type **int** consists of all integers, together with operations '=', '~=', '<', '=<', '>', '>=', '+', '-', '*', '/', '↑', and MOD provided without formal definition, i.e., they are "known." Integers are mathematical entities; there is no upper limit on the magnitude that may be expressed. Since integers are of great utility in hardware description languages, but not hardware objects, many functions are assumed in int. An integer is denoted with a contiguous sequence of symbols, digits and capitals that may be partitioned into the sign part, magnitude part and base indicator. The sign part consists of symbol + (optional) or symbol -. The magnitude may be expressed in decimal, binary, octal, or hexadecimal using the digits (and capitals) appropriate to the base indicator (decimal-none, binary-B, octal-O, and hexadecimal-H).

Type **bool** has two members, denoted by 1 and 0 representing "true" and "false" respectively, together with operations '=', '~=', '&', '|', '~', '<', '=<', '>', '>='. A small, but not minimal, set of familiar, basic operations is assumed here and in the types below to ensure the semantics of the majority of hardware oriented operations by requiring that they be built. The choice of '&' for AND, and '|' for OR is consistent with some languages and notations, but certainly not all hardware description languages. The more desirable symbols $\wedge$

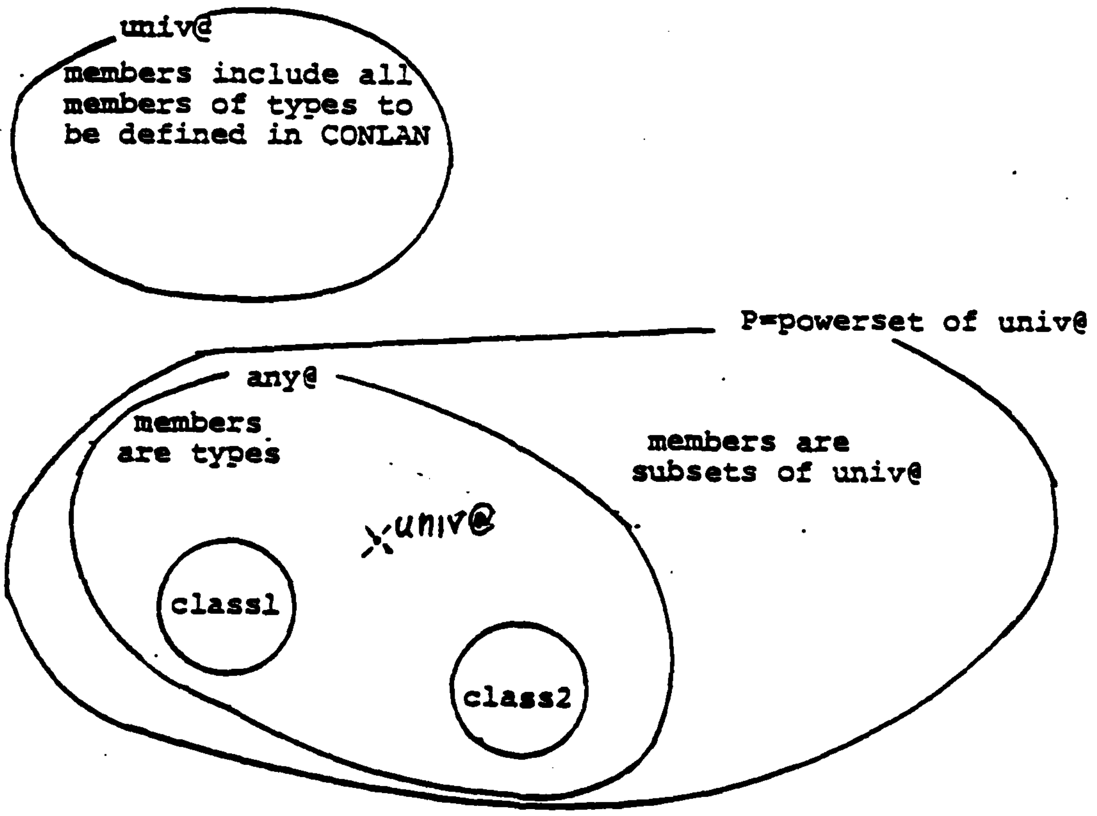

univ@
members include all members of types to be defined in CONLAN
P=powerset of univ@
any@
members are types
members are subsets of univ@
univ@
class1
class2

Figure 1.3-1 Univ@ and Any@

and V are provided in the publication character set. The relational operators are based upon '0' being less than '1'.

Type string consists of all sequences of characters, together with operations '=', '~=', '<', '=<', '>', '>=', and order@. A string is denoted by enclosing the sequence in single quotes ('). The relational operators are based on the order of characters in the ISO-646 table. Further, function order@ returns an integer that is unique to the string given as a parameter. Another useful order on strings is established through these integers.

Type tuple@ consists of all lists of elements of univ@, together with operations '=', '~=', size@, select@, remove@, and extend@. Tuple@ includes the empty list. A tuple is denoted via a list of object denotations enclosed in '(.' and '.)', and separated by commas. Two tuples are equal ('=') if they have the same number of and identical members in identical order. Otherwise they are not equal ('~='). Function size@(x) returns the number of members of a tuple. Consecutive integers from 1 to size@(x) identify the positions of the members of tuple x. Only the positions of this range may be referenced without error. Function select@(x, i) returns the member of tuple x in position i. Function remove@(x,i) returns the tuple derived from x by deleting its ith member. Function extend@(x,u) returns the tuple derived from x by inserting u as a new, last member.

Type cell@(t:any@) comprises an infinite number of elements each of which may hold at most one element of type t at any point in time. This element is called the cell's content, and may change over time. Cells are therefore the basic carriers of CONLAN. No constant denotation exists for cells. A cell must be assigned a name via a declaration statement before it can be distinguished from another cell, just as a description must be instanciated. Function cell_type@(x) returns the type of the (potential) contents of cell x. Function empty@(x) tells if cell x is empty or not. Function get@(x) returns the contents of cell x. If the cell x is empty an error condition exists. Activity put@(x,u), the primitive activity of CONLAN, replaces the contents of cell x with u. Cells are modifiable objects in that their contents may change over time. These are the only pscl objects with this attribute and constitute the bases for all modifiable objects in the CONLAN family.

It took the Working Group a great deal of time to refine this minimal collection of types and a class. We believe that it was time well spent, however, and that all useful hardware description types can be derived from this fundamental collection.

1.4 Base Conlan (bcl) Preview

Bcl was developed to illustrate and test the language constructions mechanism and to provide the types expected to pervade the CONLAN family. While the Working Group has no control over toolmakers, we feel that derived languages are truly 'CONLAN members' if they are derived from bcl, not pscl. We expect toolmakers to adhere to our definitions of arrays, records, signals, signal carriers, etc. thus removing all justification for repeating the basic task of constructing a language from pscl and provide all the objects that they require to develope user oriented members of the CONLAN family.

The type derivation tree of bcl is shown in Figure 3. A complete discussion of all these types is out of order here, so this chapter will concentrate on motivation for and justification of the bcl types.

1.4.1. Introductory Types

Subtypes **nnint** (nonnegative integers), **pint** (positive integers) and **bint** (bounded integers) are very useful in the developement of bcl. Further, they illustrate the subtype concept. Operations on members of these subtypes are carried from parent type int and performed as if the members were integers. If results fall outside of the subsets of these subtypes, type checking on those results is responsible for detecting the error. Subtype bint provides a type family. When studying the formal construction of bcl in part II of this report, observe how bint is used. Actual parameters pick a member of the bint family, a finite range of contiguous integers, and that member is used to type an identifier.

Boolean operations **nor**, **xor**, **equ** and **nand** are included to provide users with these desirable operations and to illustrate the writing of floating operations in a CONLAN segment.

Type **tern** was included to illustrate set definition via enumeration and the construction of a finite, multivalued logic. We expect such logics to be developed; ternary logic is perhaps of greatest interest to users.

Format statements in the boolean and ternary function definitions illustrate syntax modification. Bcl syntax is being formed from pscl syntax. The modified syntax may be used in susbsequent segments; tern.lt is defined in terms of tern.leq symbolized with '=<'. That same symbolism provides the parallel operation on bool objects. This multiple use of symbolism is common in CONLAN; the types of the operands determine which specific operation is being invoked and if ambiguity exists, the function identifier must be compounded with the type identifier, i.e., tern.lt.

Typed tuples, tuples in which all members are of a specific type, are fundamental to all of the structured objects developed in bcl. The tytuple@.select function is the same as the select@ function of pscl, but that function was not carried so that the CONLAN subscripting notation, [...], could be introduced and used in subsequent definitions. Functions tytuple@.extend similarly introduces the concatenate symbol, '#', of CONLAN; function tytuple@.catenate generalizes the concept. Subtypes **inttuple** and **inttuple** provide very

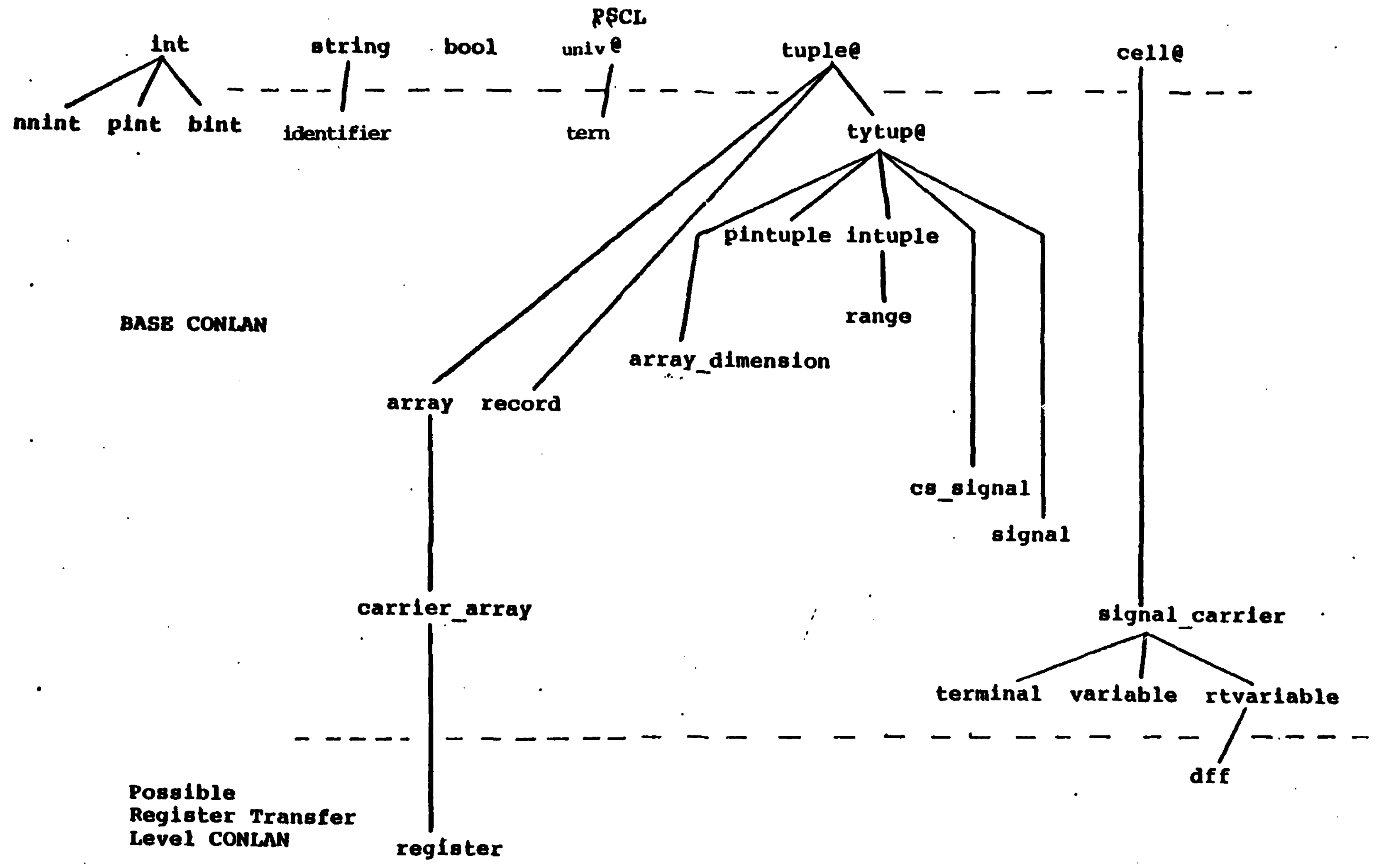


Figure 1.4-1. Type Derivation Diagram

useful, special cases of the typed tuple.

1.4.2. Arrays and Records

Types **range**, **array_dimension** and **indexer** are defined to support the definitions of arrays, records, and other structured objects of bcl. In brief, a range is a tuple of consecutive, ascending or descending integers. A format statement provides a notation for expressing a range; a colon (:) separates the left and right bounds of the list of integers. Such notation is expected to be very useful in expressing dimensions of and selecting parts of arrays. An array_dimension is a tuple of ranges. Syntax modification indicates that the ranges are to be separated by semicolons (;). Multi- dimensioned arrays are therefore supported in bcl with notation that is widely, if not universally used. Indexers are not unlike array_dimensions, but used in the selection process as opposed to the array declaration process.

In brief, a CONLAN array is a tuple with two members. The first member records the declared subscript range(s) of the array, and therefore provides the form of the array as well. The second member records the objects organized in the one or more dimension, rectangular form that usually is termed an array. Users of CONLAN arrays are not necessarily aware of this structure, of course, but the definition of functions on arrays demands that it be carefully defined.

The array functions defined in bcl might well be anticipated. Arrays may be subscripted, transposed and concatenated. In addition, to retain the uniform meaning of '=' and '~=' as given in pscl for type tuple@, additional equality testing functions are defined. Functions **equal** and **not_equal** test two arrays for identical value parts and compatible dimension parts, and functions **eq** and **neq** return arrays in which the equality or not of individual values are expressed. Further, generic operations (a parameter is an operation) are provided for arrays so that functions like int.+ to be performed on corresponding objects of arrays with the same dimensionality to form a result array of that dimensionality need not be redefined on arrays.

One of the finer points of bcl development may clarify this concept: while function xor on boolean objects is equivalent to ~= on those objects, xor was included (1) to give a prefix notation should it be preferred, and (2) so that xor can be generic as nor, etc. are. Function ~= on arrays already has a quite different meaning.

Arrays returned by array functions are always normalized, i.e., their subscript ranges begin with 1 and ascend, and degenerate dimensions are removed. In the limit, selecting one object from an array gives that object, and not an array with all the dimensions of the original array but all subscript ranges reduced to 1:1.

Records are structured objects in which the components may be of different types and are referred to by a name rather than a number as in array subscripting. A number of types are defined in bcl to support type **record**; one that may be of interest here is type **identifier**. While identifiers are used throughout CONLAN, they are not an assumed type of pscl. The first place that identifiers are required as objects of CONLAN is in records, and hence they are constructed from the more fundamental and more generally useful type 'string' of pscl at this point in bcl development. A selecting function on records permits one to extract the object in a field of the record; a format statement in the definition of the select function introduces the exclamation mark (!) separating the record and field identifiers as the CONLAN notation for field selection.

1.4.3. Values

Class value is defined to consist of all types of CONLAN equal to or derived from primitive types int, bool or string. Thus values are static, nonstructured objects, a rather traditional definition. This class is introduced to permit placing a strong restriction on the signals and carriers of CONLAN. Objects like signals of signals and carriers of carriers that have no hardware significance as far as the Working Group is able to ascertain are excluded from CONLAN, as we will see.

1.4.4. Time and Signals

There is no formal definition of time in pscl. Expressions, including predicates, are part of pscl and the process for evaluation of expressions is postulated as part of that language. 'Before-after' relations are needed in expression evaluation, but no advancement of real time is required. The CONLAN model of time is introduced where it is first required; it is superimposed on CONLAN signals after they are derived from typed tuples.

Realtime is broken into uniform durations called intervals identified with integers greater than zero. Ascending, successive integers are associated with contiguous intervals. No relation between the interval and the real time second exists in general. An implementation may impose such a relation or permit users to specify such a relation. At the beginning of each interval there are an indefinite number of calculation steps identified with integers greater than zero. Successive steps provide a before-after relation only. Real time does not advance as successive steps are taken.

The user of a low level CONLAN family member, for example gate level CONLAN, will have available a variety of activities, some of which will cause real time to advance and some not. He may take advantage of this by using computation steps to accomplish bookkeeping calculations and advancing real time only to model time dependent circuit activity. In higher level CONLAN family members, single interval activities may have disappeared to be replaced by clock-synchronized activities, each invocation of which will cause several real time intervals to elapse. At this level complex calculations will be accomplished over long sequences of computation steps with occasional clock transfers to advance real time. The precise timing interpretations of all activities and functions at any level is formally linked to bcl by activity and function segments.

The environment is assumed to support two special objects that support the CONLAN model of time: t@

is an object of type int whose value is the current time interval; s@ is an object of type int whose value is the current calcuation step. Contiguous values are provided in ascending order starting with one in both cases; therefore those objects are referred to as 'counters.' When the environment determines via an evaluation algorithm that it is appropriate, it increments the value provided by t@ and resets the s@ counter to 1.

A computation step signal, **cs_signal@(x:value)**, is a typed tuple of values of a signal at successive steps of an interval. Thus we may have computation step signals of integers, booleans, strings, ternarys, etc., but not of cells, tuples, and objects such as arrays derived from these. (We may have arrays of computation step signals, however.) The order of values in these tuples is taken to be aligned with the successive values of s@.

Functions of type cs_signal@ permit selecting one value from a computation step signal and extending such a signal (selecting a larger tuple from set cs_signal@ that is identical to a given, shorter tuple in all positions of the shorter tuple). A format statement introduces the use of braces, { and }, to select via subscripting-like notation. Toolmakers benefit; users may not use types with an @ in their identifier and in general will not be aware of the structure of CONLAN signals.

A CONLAN signal, type **signal@(x:value)**, is a tuple of computation step signals of values of type x. Each cs_signal@ of a signal@ corresponds to a value of t@; their order in a signal@ corresponds to the successive values of t@. A signal then is a tuple of tuples of values.

Functions of type signal@ permit selecting a cs_signal@ or a value from a cs_signal@, extending a signal, and the generation of a default signal from a value. The last function is needed to provide freedom in writing parameters of functions and activities, and is marked INTERPRETER@ to indicate that it is to be invoked by the environment when required to achieve type conformity in parameter passing.

Computation step signals and signals are tuples and hence may be very large, possibly infinite objects. While the use of such tuples to record histories of values is conceptually most appealing to the Working Group, and hopefully other toolmakers and users, infinite tuples clearly can't be supported and very large tuples will not lead to efficient software. The Working Group recognizes these things and does not propose that software retain such histories. We do not mean this report to be taken as an indication of how efficient software is to be written, but rather as specification of the semantics and syntax that efficient software must provide.

1.4.5. Carriers

The generic carrier of CONLAN is built in bcl as type **signal_carrier@(x:value; di:x)**. As with signals, we see that a type family is provided, but here two parameters are involved. The first indicates the type of value to be recorded in the signals carried. The second parameter provides a default or initial value for the signals. This value is used for initialization purposes and when no operation is invoked that calculates a value for a computation step.

A signal_carrier@ is a cell that carries a signal of the value type specified by the first parameter. Being a cell, a signal_carrier must be declared before it can be referenced via a name, and it is a modifiable object. The signal that it holds changes with each advance of s@ and t@. A signal is a history of values; a signal_carrier is a container of the most complete, possibly infinite history for the current s@ and t@ values. Functions of type signal_carrier@ return the default part, the signal part of a signal carrier, the present signal value (the value stored for interval t@ and step s@), and the last step value for an interval of the past. The 'present value' function is marked INTERPRETER@ and hence provides automatic dereferencing when a value is needed and the name of a carrier has been supplied. The past interval, last step value function is named delay. It may be invoked with the symbol '%' and is provided for the hardware user who will immediately recognize its importance in modeling electronic signal propagation.

Three special types of signal_carriers are built in bcl for users of bcl. Toolmakers are expected to find these types sufficient to build all future hardware description languages. Thus the Working Group does not expect toolmakers to have to go back to the generic signal_carrier@ to build the carriers of the future. The Working Group has discussed ways of preventing them from doing so, in fact, but of course it has no real way of preventing toolmakers from doing anything.

Terminals are signal_carriers with no retention properties between computation steps or intervals. Activity connect, symbolized '.=', places a value given as the right operand of .= in the s@+1st position of the current interval of the carrier named as the left operand. But if no connect activity is invoked during a computation step to extend the signal of a terminal, the default value of the terminal is placed in the signal by an interpreter activity. Another interpreter activity introduces a new cs_signal in the signal of each terminal when interval counter t@ is about to be incremented. These activities are named **finstep@** and **finint@**, respectively. The three special types of signal_carriers differ primarily in their finstep and finint activity definitions.

Boolean terminals model wires. Some wires float high and others float low when not driven. Subtypes **btm0** and **btm1** are provided with obvious fixed default values so that users are relieved of repeatedly supplying these commonly desired default values. Subtype **btml(default:bool)** permits users to specify a default value for a boolean terminal, should they perfer to do so. Subtypes **ttml(default:tern)** and **itml(default:int)** are expected to be useful, but the Working Group does not forsee a most useful default value for ternary and integer terminals.

Variables are signal_carriers with retention properties. Activity **assign**, symbolized ':=', is provided to extend the signals of variables. If assign is not invoked for a variable, activity finstep@ extends with the current value of the signal. Thus a value is carried from computation step to the next step when a new value is not provided. Finint@ initializes a new cs_signal with the last value of the current cs_signal when t@ is about to be incremented. Variables are therefore much as found in programming languages. Boolean variables may be thought to model idealized latches. Subtypes bvar(init:bool), tvar(init:tern) and ivar(init:int) are provided so that users need supply only initial values for boolean, ternary and integer variables.

Real time variables, type rtvariable(x:value; init:x), model idealized, unit delay flip-flops. Activity transfer, symbolized '<-', may be invoked to place a value in a new cs_signal of the signal of a rtvariable. Present values of signal may thus be used to compute the next interval value carried by a rtvariable. All computation step values of an interval are the same. Again, the finstep@ and finint@ activities provide the distinction between terminals, variables, and real time variables.

1.4.6. A bcl Goal

Bcl was built to provide only those objects and operations considered to be fundamental to the development of all CONLAN member languages. Much of bcl is therefore provided for the benefit of toolmakers; some of the objects and operations are clearly of immediate value to users, but bcl was never intended to be, and is not, a good hardware description language. Only time and cooperation on the part of toolmakers and users will tell if the Working Group was successful in reaching its goal of providing the base CONLAN member.

1.5 Directions

It is appropriate to comment on the status of CONLAN at the time of the initial distribution of this report. Where, at this time, is the widely acceptable hardware description language which was envisioned when the Conference on Digital Hardware Languages first met in 1973? Section 1.1.1 should have convinced the reader that a family of languages rather than a single language is required. Base CONLAN, as developed in part II of this report, is a formal foundation for this family of languages. Not only is language development from a base language the soundest approach, but it is the approach most likely to gain acceptance by the community of language designers. It can be said with some confidence that if there are to be a widely accepted,widely used hardware description languages, they will be developed from Base CONLAN.

To promote an orderly development of hardware description languages and to enhance their acceptance in an industrial environment, a powerful construction mechanism for such languages, based on a common core syntax was developed. This construction mechanism ensures that the semantics of the languages derived are well defined. Further, semantically related languages can be constructed which permit the description of digital systems at different levels of abtraction. The common core syntax facilitates learning a new language written in the CONLAN framework. In addition, capabilities for syntax modification permit the suppression of unneeded constructs and the introduction of shorthand for frequently used objects to obtain simple yet useful languages. We feel that the primitive set language, together with the construction mechanism are not only valid for the development of hardware description languages, but also represent a contribution to the available techniques for formal semantic specification of operative languages, including programming languages.

Base CONLAN is primarily a starting point, with well-defined and semantically sound primitives, for language designers to derive a coherent and comprehensive family of digital system description languages. More time and effort will be needed before user oriented, less general but simpler, special purpose languages will be developed. The Working Group has verified the completeness of Base CONLAN with the derivation of portions of several simple languages above the bcl level but is stopping short of presenting to the public any user level languages at this time. Timeliness in the distribution of this report is one reason for the absence of user level languages, but more importantly, the Working Group now seeks a broad spectrum of response from the community of users and language designers. It is expected that some user level languages will become standards for purposes of communication throughout the computer establishment. It would be a mistake for the Working Group to publish languages which might be misinterpreted as proposed standards, but at the same time be regarded by the user community as, for some reason, inadequate.

The development of languages from bcl by the Working Group has been hampered by constant fluctuation in bcl itself. Finally, this language has reached a steady state. With the publication of this report members of the Working Group and other toolmakers may regard Base CONLAN as fixed. The development of user level languages may now proceed without fear that they will become obsolete due to changes in the postulated base language.

References

1. Piloty, R., Barbacci, M., Borrione, D., Dietmeyer, D., Hill, F. and Skelly, P., "CONLAN -- A Formal Construction Method for Hardware Description Languages: Language Derivation," Proceedings National Computer Conference, Volume 49, Anaheim, California, 1980.

2. Piloty, R., Barbacci, M., Borrione, D., Dietmeyer, D., Hill, F. and Skelly, P., "CONLAN -- A Formal Construction Method for Hardware Description Languages: Language Application," Proceedings National Computer Conference, Volume 49, Anaheim, California, 1980.

3. Barbacci, M. R., "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems," IEEE Computer Society, Transactions on Computers, Volume C-24, Number 2, February 1975.

4. Piloty, R., "Guidelines for a Computer Hardware Description Consensus Language" (2nd draft), Memorandum to the Conference on Digital Hardware Languages, June 6, 1976.

5. Liskov, B. and Zilles, S., "Programming with Abstract Data Types," SIGPLAN Notices 9, pp. 50-59, April 1974.

6. Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C., "Abstraction Mechanism in CLU," Computation Structures Group, Memo 144-1, MIT January 1977.

7. Wulf, W. A., "Alphard: Toward a Language to Support Structured Programming," Technical Report, Department of Computer Science, Carnegie-Mellon University, April, 1974.

8. Wulf, W. A., London, R. L. and Shaw, M., "Abstaction and Verification in ALPHARD," Technical Report, Department of Computer Science, Carnegie-Mellon University, March 1976.

9. Lucas, P. and Walk, K., "On the Formal Description of PL/1." Annual Review of Automatic Programming, Vol. 6, part 3, 1969.

10. Jacquet, P., "Les Types Generic: Proposition pour un Mecanisme d'Abstraction dans les Langages de Programmation."

APPENDIX A

CONLAN Basic Symbols

CONLAN is based upon the characters of the International Standard 7 Bit Coded Character Set for Information Processing Interchange (ISO 646) International Reference Version. In addition, an extended set of symbols may be used in CONLAN descriptions intended for publication only.

Code	Character	Semantics
00-1F		control characters (treated as space)
20	space	terminator
21	!	record access
22	"	part of comment delimiter
23	#	catenation
24		escape character
25	%	delay
26	&	logic AND
27	'	string delimiter
28	(	expression, tuple, and
29	)	parameter list delimiter
2A	*	multiply
2B	+	positive, add
2C	,	list separator
2D	-	negative, subtract
2E	.	compound identifiers, tuple denotation
2F	/	divide, part of comment delimiter
30-39	0 1 2 3 4 5 6 7 8 9	digit
3A	:	typing separator, subscript range, label
3B	;	list separator
3C	<	less than
3D	=	equal
3E	>	greater than
3F	?	(available)
40	@	used in system identifiers
41-5A	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z	capital
5B	[	subscript header
5C	\	(available)
5D	]	subscript terminator
5E	↑	power
5F	_	(underscore) used as a letter in identifiers
60	`	(available)
61-7A	a b c d e f g h i j k l m n o p q r s t u v w x y z	letter
7B	{	enumerated set, time reference
7C	|	logic OR
7D	}	enumerated set, time reference
7E	~	logic NOT
7F	delete	treated as space

Table 2.1-1: ISO 646-IRV Characters and CONLAN Semantics

2.1.1. ISO 646-IRV Character Assignments

Table 2.1-1 provides the ISO 646-IRV character set and codes (hexadecimal), and the CONLAN semantics of most characters. The character set is partitioned into (1) control characters, (2) capitals and letters, (3) digits, (4) space, and (5) symbols.

ISO 646-IRV codes 00 through 1F, and code 7F are treated as spaces by CONLAN implementations. The space (code 20, as well as codes 00-1F and 7F) may be used freely to enhance the readability of a CONLAN description. Spaces may not appear within compound symbols (Sec. 2.1.2), keywords (Sec. 2.1.5) or identifiers (Sec. 2.1.6).

Elements of a list are separated by comma (code 2C) and semicolon (code 3B).

The semantics of symbols (?), (\), and (`) (codes 3F, 5C, and 60, respectively) may be assigned by CONLAN toolmakers.

It is the responsibility of the toolmaker to provide the full ISO 646-IRV character set even when equipment to support an implementation does not. The international monetary symbol, (, code 24) must be used to accomplish this. For example, if equipment provides only upper case letters, " u" may be used to signify "shift to upper case" and " l" to signify "shift to lower case".

Equipment may provide characters not listed in Table 2.1-1 in place of characters shown. An equipment dependent character may appear in CONLAN documents as a replacement for an ISO 646-IRV character with the same code. For example, the following substitution is made in this document:

$ for

2.1.2. CONLAN Compound Symbols

Table 2.1-2 depicts the CONLAN compound symbols and their semantics.

Other compound symbols may be assigned meaning by CONLAN toolmakers. For example, <= , -> , and => are expected to appear in working languages developed from bcl.

2.1.3. Publication Symbols

CONLAN documents destined for publication only may: (a) underline keywords or set them in bold face, and (b) replace implementation symbols as shown in Table 2.1-3.

Symbol	Meaning
>=	greater than or equal
=<	less than or equal
~=	not equal
.<	set member
*/	comment head
/*	comment terminator
(.	tuple denotation head
.)	tuple denotation terminator
<|	derived from
.=	terminal connection (bcl)
:=	variable assignment (bcl)
<-	transfer (bcl)

Table 2.1-2: CONLAN Compound Symbols

∨ for |	logic OR
∧ for &	logic AND
¬ for ~	logic NOT
≥ for >=	greater than or equal
≤ for =<	less than or equal
≠ for ~=	not equal
∈ for .<	set member
Δ for %	delay

Table 2.1-3: PublicationSymbols

2.1.4. The @ Symbol

Certain keywords and identifiers whose use is restricted to the definition of languages are tagged by the presence of the symbol @ as the last character of the keyword or identifier.

Keywords and identifiers terminating with this symbol can appear within the scope of a language definition segment. Their use is therefore limited to toolmakers.

2.1.5. Keywords

Keywords are sequences of capitals (Table 2.1-1) possibly terminated by symbol @. The character immediately following a keyword must be a space or a symbol. Keywords ending with symbol @ may be used in language definitions only, i.e. within CONLAN segments (2.4.7) only.

All sequences of capitals, letters, digits, underscore (_), and symbol @ that begin with "END" are interpreted as keyword END. Recommended statement termination keywords are suggested in the pertinent sections.

The sequences of capitals and symbol @ listed in Table 2.1-4 are keywords of all members of the CONLAN family.

ACTIVITY ALL ASSERT ATT
BODY
CARRY CARRYALL CASE CONLAN
DECLARE DESCRIPTION
ELIF ELSE END EXTEND EXTERNAL
FORALL@ FORMAT@ FORONE@ FORSOME@ FUNCTION FROM
IF IS INTERPRETER@ IN INOUT
MEANS
OUT OVER
PRIVATE
REFLAN REPEAT RETURN REMOVE
STEP
THE@ THEN TO TYPE
USE
W WITH

Table 2.1-4: CONLAN Family Keywords

2.1.6. Identifiers

CONLAN entities, such as types, elements of types, languages, descriptions, operations are generally referenced via identifiers starting with at least one letter followed by an arbitrary string of letters , digits, and symbol underscore (_) (Table 2.1-1) of any length and terminating with a letter or digit (user identifiers) or with the symbol @ (system identifiers). The character immediately following an identifier must be a space or a symbol.

Certain primitive operations are provided with an infix format in which the operation is referenced through a symbol or a symbol string instead of an identifier (e.g. +). Toolmakers may provide, via FORMAT@ statements (Sec. 2.5.4) prefix and infix symbol strings or identifiers to denote new operations.

Compound identifiers are two or more identifiers separated by the symbol period (.).

A CONLAN implementation may place a length limit greater than or equal to six (6) on identifiers other than system identifiers.

Toolmakers may use symbol @ as the last character of identifiers used only in language construction. Users may not use such identifiers and will generally not be aware of their existence.

Primitive Set CONLAN (pscl)

CONLAN is built from the primitive set language pscl defined here. Pscl is an exceptional member of the CONLAN family in that:

1. there is no reference language for pscl,

2. object sets are assumed rather than defined, and

3. operators on objects are assumed rather than being defined in more fundamental terms.

2.2.1. Objects

"Objects" are the things with which CONLAN is concerned. Pscl provides simple objects such as integers, strings, Booleans, etc. and lists of objects ("tuples") as the basic structured objects. Bcl provides more elaborate structured objects derived from tuples: carriers, signals and arrays. Other members of the CONLAN family are expected to provide gates, registers, microprocessors, etc. as objects. Objects are collected into sets which may or may not be explicitly named.

All objects are members of at least one named set (univ@) and are usually also members of named subsets of univ@. Named sets of objects are brought into existence via TYPE segments and are members of a named set, any@.

2.2.2. Types and Classes

A "type" is a named set of objects together with operations defined on the members of that set. Language constructs are provided for defining new types (Sec. 2.4.4). Types may be defined in terms of a formal parameter or parameters. Such definitions provide a "family." A member of the family is obtained by binding the formal parameters with actual parameters.

It will be necessary to group together a set of types which may have a common property. Such a set of types will be called a "class". It is important to note that members of a class are types, not the elements of these types. Only one class, any@, which will contain all types to be defined in any CONLAN family member is included in pscl. A constructive method (Section 2.4.6) is provided to define additional classes.

2.2.3. Operations

An "operation" is a formal rule for (i) selecting one or more "result" objects, each from a designated result type, or (ii) modifying the contents of one or more container objects. The rule is usually dependent upon one or more supplied representative objects known as "operands" or "parameters", each of a designated type.

The parameter types define the domain of the operation. The result types define the range of the operation.

The evaluation of the formal rule can be specified in terms of already known operators. Operators whose formal rule is implied are called "primitive". CONLAN provides functions and activities (Sec. 2.4.1) as categories of operations.

2.2.4. Strong Typing

Experience with programming languages has revealed the value of requiring users of a language to specify the set of which each referenced object is a member, i.e., to specify the type of all identifiers used. While hardware description languages have not provided traditional data types (integer, real, complex, etc.), many have introduced hardware types (register, terminal, clock, switch, etc.). To gain error checking capability CONLAN requires that the type of all objects be specified via designators.

A designator is an identifier of a type or class. Designators are found in formal parameter lists, predicate forms, set constructors, instantiation of descriptions, and declarations.

There are two categories of designators:

1. non-generic designators identify a specific DESCRIPTION or TYPE segment

```
description_identifier
description_identifier(attributes)
type_identifier
type_identifier(actual_parameters)
```

2. generic designators identify a class of segments

```
any@                                            any TYPE segment
class_identifier(actual_parameters)
ACTIVITY                                        any ACTIVITY segment
ACTIVITY(actual_parameters)
FUNCTION . type_designator                      any FUNCTION segment
FUNCTION(actual_parameters) : type_designator
DESCRIPTION                                     any DESCRIPTION segment
DESCRIPTION(attributes)
```

The set to which an object belongs is denoted using the following formats:

```
representative : designator
representative [ dimensions ] : designator
```

2.2.5. Pscl Object Types and Operations

Table 2.2-1 depicts the types and operations which exist in pscl. The table shows, for each type, the domain and range of its operations.

TYPE	OPERATION(S)	DOMAIN	RANGE
univ@	= ~=	univ@ x univ@	bool
any@	= ~= <|	any@ x any@	bool
	.<	univ@ x any@	bool
int	= ~= < =< > >=	int x int	bool
	+ - * / ↑ MOD	int x int	int
bool	= ~= < =< > >=	bool x bool	bool
	&|	bool x bool	bool
	~	bool	bool
string	= ~= < =< > >=	string x string	bool
	order@	string	int
tuple@	= ~=	tuple@ x tuple@	bool
	size@	tuple@	int
	select@	tuple@ x int	univ@
	extend@	tuple@ x univ@	tuple@
	remove@	tuple@ x int	tuple@
cell@	get@	cell@	univ@
	put@	cell@ x univ@	cell@
	cell_type@	cell@	any@
	empty@	cell@	bool

Table 2.2-1: Pscl Types and Operations

2.2.5.1. univ@

Type univ@ consists of all members of all types defined or yet to be defined in all members of the CONLAN family, together with operators '=' and '~='. It permits the present definition of operations on objects to be defined in the future. The pscl universe is, in part[1]:

univ@={.., -1, 0, +1, .., '!', .., 0, (.0, 0.), ..,'xYz', ..}

Type univ@ is considered as the defining type for the other types of pscl, namely "int", "bool", "string", "cell@", "tuple@" from which all other types of CONLAN will be derived.

[1]Members of cell@ have no constant denotation and therefore can not be illustrated

2.2.5.2. any@

Class **any@** is the universal class in CONLAN, and the only class known in pscl. Its domain is the set of designators for all types defined or yet to be defined in all members of the CONLAN family, together with operations '=', '~=', '.<', and '<|'. At this point in the development of CONLAN:

any@ = {univ@, int, bool, tuple@, cell@, string}

Function '.<' may be used to determine if an object from univ@ is a member of a defined type (a member of any@).

Function '<|' may be used to determine if a member of any@ (a type) was derived from another member of any@.

2.2.5.3. int

Type **int** consists of all integers, together with a substantial number of operators provided without formal definition, i.e., they are "known." An integer is denoted with a contiguous sequence of symbols, digits and capitals that may be partitioned into the sign part, magnitude part and base indicator. The sign part consists of symbol + (optional) or symbol - . The magnitude may be expressed in decimal, binary, octal, or hexadecimal using the digits (and capitals) appropriate to the base indicator as shown in table 2.2-2.

NUMBER SYSTEM	BASE INDICATOR	DIGITS AND CAPITALS AVAILABLE TO EXPRESS MAGNITUDE
decimal	none	0 1 2 3 4 5 6 7 8 9
binary	B	0 1
octal	O	0 1 2 3 4 5 6 7
hexadecimal	H	0 1 2 3 4 5 6 7 8 9 A B C D E F

Table 2.2-2: Integer Denotation

Example: -12 = -1100B = -14O = -CH

2.2.5.4. bool

Type **bool** has two members, denoted by 1 and 0 representing "true" and "false" respectively, together with operations '=', '~=', '&', '|', '~', '<', '=<', '>', '>='. The relational operators are based upon '0' being less than '1'.

2.2.5.5. string

Type **string** consists of all sequences of characters, together with operations '=', '~=', '<', '=<', '>', '>=', and order@. The objects of string are denoted by enclosing the sequence in single quotes ('). Sequences such as '1A', 'b + 5', 'himmelherrgottsakrament ah tchoum !!!, said the white rabbit' are included. The character ' must be doubled if it is to appear in a string denotation (e.g. 'What''s his name?').

The relational operators are based in the order of characters in the ISO-646 table. When comparing strings of different length, the shorter string is extended with trailing spaces (code 20) to equalize the lengths.

Function order@ takes a string as parameter and returns an integer computed by treating the elements of the strings (i.e. the characters) as 'digits' in a base 128 representation. The leading character is the most significant 'digit'. For instance, order@('Xy2') returns (58H*128*128 + 79H*128 + 32H), that is, 163CB2H.

2.2.5.6. tuple@

Type **tuple@** consists of all lists of elements of univ@, together with operations '=', '~=', size@, select@, remove@, and extend@. Tuple@ includes the empty list.

A tuple is denoted via a list of object denotations enclosed in '(.' and '.)', and separated by commas.

Two tuples are equal ('=') if they have the same size and identical members in identical order. Otherwise they are not equal ('~=').

Function size@(x) returns the number of members of a tuple. If the tuple is empty, size@ returns 0. Consecutive integers from 1 to size@(x) identify the positions of the members of tuple x. Only the positions of this range may be referenced. Attempts to reference positions outside this range result in an error report.

Function select@(x, i) returns the member of tuple x in position i (if integer i is in the range 1 through size@(x)).

Function remove@(x,i) returns the tuple y such that y holds all components of x in the same order except the ith one if i is in the range from 1 to size@(x). If i is ouside the range, an error condition. If size@(x) = 1 then the empty tuple is returned.

Function extend@(x,u) returns the tuple y of size@(x)+1 such that the components of y are identical to those of x in its leftmost size@(x) positions and that the component in position size@(x)+1 is equal to u.

Tuples are never modified. Remove@ and extend@ simply select a member of the set of all tuples with the right size and contents. The original tuple is not altered.

2.2.5.7. cell@

Type cell@(t:any@) comprises an infinite number of elements each of which may hold at most one element of type t at any point in time. This element is called its content. The content may change over time[2].

No denotation exists for cells. A cell must be named in a declaration statement before it can be referenced in an operation. Cells are initially empty.

Function cell_type@(x) returns a member of any@, namely the type of the (potential) contents of cell x.

Function empty@(x) returns 1 ('true') if cell x is empty otherwise it returns 0 ('false').

Function get@(x) returns the contents of cell x. If the cell x is empty an error condition exists.

Activity put@(x,u) replaces the content of cell x with u. The type of u must be the specified content type of cell x. An attempt to put an element of the wrong type result in an error. If cell x is empty, put@ simply inserts u in the cell.

Cells are potentially modifiable objects (via function put@). These are the only pscl objects with this property and constitute the bases for the development of carriers, variables, and other modifiable objects in the CONLAN family.

2.2.6. Operator Precedence

In an expression that invokes symbolized (infix) operators, where parentheses do not dictate the order of invocation, operators are invoked in order of decreasing precedence numbers. There are ten levels of operator precedence in CONLAN, as shown in table 2.2-3.

Operators with the same precedence are invoked in the left to right order of their appearance. Operators may be added or removed via FORMAT@ statements (Sec. 2.5.4). The precedence of the symbols given in the table above is fixed throughout CONLAN: An operator can be removed, but if reinserted, it must have the same precedence number.

[2] There is not concept of time in pscl. It will appear in the development of bcl.

Precedence	Operators	
1	.= <- :=	(connection, assignment, transfer)
2	|	(disjunction)
4	&	(conjunction)
5	= ~= < =< > >= .< ◁	(relationals, member, derived)
6	+ -	(binary arithmetic)
7	* / MOD	(binary arithmetic)
8	↑	(power)
9	~ - +	(negation, unary arithmetic)
10	#	(catenation)

Table 2.2-3: Operator Precedence

Appendix C

CONLAN Block Structure and Rules

CONLAN text consists of properly nested blocks. Blocks which define an entity such as a type, a language, a hardware module or an operation, are called "segments." Blocks which refer to or invoke an instance of a segment such as declarations, operation invocations, etc. are called "statements." Segments begin with a keyword followed by an identifier and terminate with keyword END, which may, but need not be followed by the segment identifier. Most statements begin with a keyword and terminate with END or ENDxxx where xxx reflects the opening keyword. There is one exception to this rule: "Operation Invocaton" is not bracketed by keywords.

Three broad classes of objects are of primary concern in working members of the CONLAN family. "Values" are static objects; they do not change with time. An integer, a character, etc. are values. "Signals" are tuples that have values as components. A different time is associated with each value. A signal is then a history of values. Signals are formally defined in Chapter 2.8 in the CONLAN report to be published as part of Base CONLAN (BCL). "Carriers" are named objects derived from primitive type cell@. These objects are formally constructed in Chapter 2.8 of the CONLAN report. For the purposes of this chapter, assume that a carrier is a cell@ whose contents (a signal) can be replaced as a result of or during an operation invocation.

2.3.1. Segments and Statements

The segments, the sections in which they are presented, and their principal intended application are:

FUNCTION	Defined operations that return an element of a type and has no side effects. The keyword FUNCTION, when used as a designator, stands for the set of all functions.
ACTIVITY	Define operations have side effects by modification of the contents of carriers named in the formal parameter list. The keyword ACTIVITY, when used as a designator, stands for the set of all activities.

DESCRIPTION	Model digital hardware. The keyword DESCRIPTION, when used as a designator, stands for the set of all descriptions.
TYPE	Define new types. The reserved identifier any@, when used as a designator, stands for the set of all types.
SUBTYPE	Denote a subset of a type or a member of a family.
CONLAN	Define new working languages.
EXTERNAL	Copy a segment definition from an external segment library (Sec. EXTERNAL!STATEMENTS).

CONLAN Block Structure and Rules

The statements, the sections in which they are presented, and their principal intended application are:

FORALL@	Test if a predicate is true for all members of a set.
FORSOME@	Test if a predicate is true for at least one member of a set.
FORONE@	Test if a predicate is true for exactly one member of a set.
THE@	Select the member of a set for which a predicate is true.
"Set enumeration"	Denotes an unordered set of objects.
ALL	Selects all members of a set for which a predicate is true.
"Direct invocation"	Function and Activity invocation.
IF	Evaluate one of several operations depending on several predicates.
CASE	Evaluate one of several operations depending on the value of a conditional or selection expression.
OVER	Repeat an operation over members of a set.
RETURN	Evaluates the result of a function invocation.
REFLAN	Define the reference language for a segment.
DECLARE	Declare an element of a type.

USE	Name an instance of a description.
FORMAT@	Define a syntax extension.
CARRY	Bring selected objects from other segments.
CARRYALL	Bring all objects from other segments.
ASSERT	Predicates that result in error reports if they evaluate to false.

2.3.2. Reference Languages

Some segments require that a "reference language" be stated. A reference language specifies an existing language used as a base for the definition of a new segment (languages themselves are defined via CONLAN segments. A reference language brings via their names functions, activities, types, and descriptions to inner segments. Such global names provide a starting point for forming the body of segments. Pscl is the reference language for bcl. Thus integers, characters, etc. and operations upon them are available for use in defining bcl.

CONLAN Environment

A CONLAN document has significance only if it is read by a person or machine. That reader (environment) is required to use available facilities to respond to and interact with the document. It must provide the type checking mechanism. It must record the names of defined and declared items and provide the data base they require. It must record signal values. From such records, it can determine facts of importance to continued document evaluation. "System interfaces" are prescribed environment responses, not necessarily defined via CONLAN syntax.

2.6.1. CONLAN Model of Time

CONLAN provides a discrete model of continuous, real time.

1. Real time is broken into uniform durations called "intervals" identified with integers greater than zero. Ascending, successive integers are associated with contiguous intervals. No relation between the interval and the real time second exists in general. An implementation may impose such a relation or permit users to specify such a relation.

2. At the beginning of each interval there are an indefinite number of calculation "steps" identified with integers greater than zero. Successive steps provide a before/after relation only.

3. Values obtained at the last step of computation are the values associated with the interval.

When modeling a specific digital system satisfactory results are obtained at reasonable computational cost by quantizing time to some fraction of the second; for purposes of example assume the nanosecond. Actual binary signals are constrained by this quantization to change at the boundaries of 1 ns. durations. Computing the value of a specific signal during a specific 1 ns. duration may require successive calculations: if a wire is driven by a gate network modelled by a&b | c&d, then a&b and c&d must be evaluated before the signal value is determined. The CONLAN interval and step support this model of digital hardware and method of simulation.

No real time is thought to elapse when evaluating a mathematical function or executing a computer program. Yet many successive computational steps are usually required. Again the CONLAN model of time supports such computation; the interval is of less importance in these cases as its identifying integer does not advance.

2.6.2. System Interfaces

2.6.2.1. error@

When this primitive activity is invoked, the processor of the document issues an error message. Subsequent actions are determined by the sophistication of the environment. All processing might stop, or if the nature of the error is determined, a default value may be returned and processing continued timidly.

2.6.2.2. t@ and s@

t@ is an object of type int whose value is the current time interval. Contiguous values are provided in ascending order starting with one.

s@ is an object of type int whose value is the current calculation step. Contiguous values are provided in ascending order, starting with one.

When the environment determines that all signals have attained stable values, it increments the value provided by t@ and resets the s@ counter to 1. It detects calculation step oscillation (s@ reaches a predetermined limit) and responds to it with a message and optionally termination of document evaluation or continuation using the signal values available at the last step of calculation.

2.6.2.3. pack@

When this activity is invoked, the environment converts a value into a signal and then inserts the signal into a carrier. pack@ is an activity which is provided by the language designer. It describes parameter type conversion operations (Sec. 2.3.4.1). The definition of pack@ must be prefixed with keyword INTERPRETER@.

2.6.2.4. content@

When this function is invoked, the environment extracts a signal from a carrier and then selects the last value in the signal. content@ is a function which is provided by the language designer. It describes parameter type conversion operations (Sec. 2.3.4.1). The definition of content@ must be prefixed with keyword INTERPRETER@.

2.6.2.5. finstep@

When this activity is invoked, the environment extends a signal one step. finstep@ is an activity which is provided by the language designer. It describes the default signal growth mechanism for a time step. The definition of finstep@ must be prefixed with keyword INTERPRETER@.

2.6.2.6. finint@

When this activity is invoked, the environment extends a signal one interval. finint@ is an activity which is provided by the language designer. It describes the default signal growth mechanism for a time interval. The definition of finint@ must be prefixed with keyword INTERPRETER@.

2.6.3. CONLAN Model of Computation

Hardware descriptions record how the signal parts of some carriers are related to those of carriers with known signal parts such as constants in a manner that displays behavior and/or organization and supports computation of those unknown signal parts. Such computation is usually performed viewing past and present signal values as "known" and future values as "unknown." With each computational step, known values are used to determine a future value and thereby change its status to known.

CONLAN signals gain a component with each calculation step and real time interval. Activities finstep@ *and finint@ provide default signal growth for a step or interval in which no user invoked* activity extends a signal. These activities are invoked automatically by the environment, and not by CONLAN users or toolmakers.

Base-CONLAN (bcl) is constructed in Chapter 2.7 from pscl so as to support this model of computation. A step of signal computation at the base level of descriptive abstraction corresponds to a computation step and consists of the following stages:

Stage	Description
1	For each invoked activity and function of the system description under evaluation, determine via the description of that invoked module the value part(s) of carriers for a future step or steps from known present and past signal values. Advance to stage 2.
2	Determine via finstep@ activities on carriers which have not been serviced in stage 1 and provide for them the missing step value according to the rule specified in the activity, future step values from known present and past signal values. Advance to stage 3.
3	Examine the record of *present and next* step values. If *one or more* carriers have differing values and s@ is less than a predetermined limit, advance the step counter s@ and return to stage 1. If s@ equals the predetermined limit, publish

an "oscillation" error message and (optionally) continue with stage 4; otherwise continue with stage 4.

4 Determine via finint@ activities on carriers, future interval values from known present and past signal values. Advance to stage 5.

5 Reset s@ to 1, increment t@, and return to stage 1.

None, one or more functions and activities may be invoked in a step for a specific carrier. If multiple invocations attempt to set a signal to different values, a "collision" exists and will be reported as an error.

ATE

LME